

Chapter 2

Background & State of the art

This chapter discusses the state of the art in the areas of *WebAssembly* and *Software Diversification*. In Section 2.1 we discuss the WebAssembly language, its motivation, how WebAssembly binaries are generated, the language specification, and security-related issues. In Section 2.2, we present a summary of Software Diversification, its foundational concepts and highlighted related works. We select the discussed works by their novelty, critical insights, and representativeness of their techniques. In Section 2.3, we finalize the chapter by stating our novel contributions and comparing them against state-of-the-art related works.

2.1 WebAssembly overview

JavaScript is currently used in all modern web browsers to allow client-side scripting. However, due to the complexity of this language and to gain in performance and its security flaws, several alternatives appeared. For example, Java applets were introduced on web pages late in the 90's to execute Java bytecode in the client side [40]. Similarly, Microsoft made two attempts with ActiveX in 1996 [78], and with Silverlight in 2007 [67]. All these attempts failed to persist or had low adoption, mainly due to security issues and the lack of consensus on the community of browser vendors.

In 2014, Alon Zakai and colleagues proposed the Emscripten tool [51]. Emscripten used a strict subset of JavaScript, `asm.js`, to allow low level code such as C to be compiled to JavaScript. `Asm.js` was first announced as an LLVM backend [52]. This approach came with the benefits of having all the ahead-of-time optimizations from LLVM, gaining in performance on browser clients [49] compared to standard JavaScript code. The main reason why `asm.js` is faster, is that it limits the language features to those that can be optimized in the LLVM pipeline. Besides, it removes the majority of the dynamic characteristics of the language, limiting it to numerical types, top-level functions, and one large array in the memory directly accessed as raw data. Since `asm.js` is a subset of JavaScript it was compatible

with all engines at that moment. Asm.js demonstrated that client-code could be improved with the right language design and standardization. The work of Van Es et al. [42] proposed to shrink JavaScript to asm.js in a source-to-source strategy, closing the cycle and extending the fact that asm.js was mainly a compilation target for C/C++ code. Moreover, JavaScript faces several limitations related to the characteristics of the language. For example, any JavaScript engine requires the parsing and the recompilation of the JavaScript code which implies a significant overhead. Consequently, the asm.js initiative, the W3C publicly announced the WebAssembly (Wasm) language in 2015. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially consolidated later by the work of Haas et al. [41] in 2017. The announcement of WebAssembly marked the first step into the standardization of bytecode in the web environment. Wasm is designed to be fast, portable, self-contained and secure, and it outperforms asm.js [41]. Since 2017, the adoption of WebAssembly keeps growing. For example; Adobe, announced a full online version of Photoshop¹ written in WebAssembly; game companies moved their development from JavaScript to Wasm Wasmlike is the case of a full Minecraft version ²; and the case of Blazor ³, a .Net virtual machine implemented in Wasm, able to execute C# code in the browser.

2.1.1 From source to Wasm

All WebAssembly programs are compiled ahead-of-time from source languages. LLVM includes Wasm as a backend since version 7.1.0 released in May 2019⁴, supporting a broad range of frontend languages such as C/C++, Rust, Go or AssemblyScript⁵. The resulting binary works similarly to a traditional shared library, it includes instruction codes, symbols and exported functions. In Figure 2.1, we illustrate the workflow from the creation of Wasm binaries to their execution in the browser. The process starts by compiling the source code program to Wasm (Step ①). This step includes ahead-of-time optimizations such as optimizations in the LLVM toolchain.

The step ② builds the standard library for Wasm usually as JavaScript code. This code includes the external functions that the Wasm binary needs for its execution inside the host engine. For example, the functions to interact with the DOM of the HTML page are imported in the Wasm binary during its call from the JavaScript code. The standard library can be manually written, however, compilers like Emscripten, Rust and Binaryen can generate it automatically, making this process completely transparent to developers.

¹<https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecAOK4V8R69lw>

²<https://satoshinm.github.io/NetCraft/>

³<https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor>

⁴<https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0>

⁵subset of the TypeScript language

Finally, the third step (Step ③), includes the compilation and execution of the client-side code. Most of the browser engines compile either the Wasm and JavaScript codes to machine code. In the case of JavaScript, this process involves JIT and hot code replacement during runtime. For Wasm, since it is closer to machine code, and it is already optimized, this process is a one-to-one mapping. For instance, in the case of V8, the compilation process only applies simple and fast optimizations such as constant folding and dead code removal. Once V8 completes the compilation process, the generated machine code for Wasm does not change anymore and is the same used along all its executions. This analysis was validated by conversations with the V8's dev team and by experimental studies in our previous contributions [10].

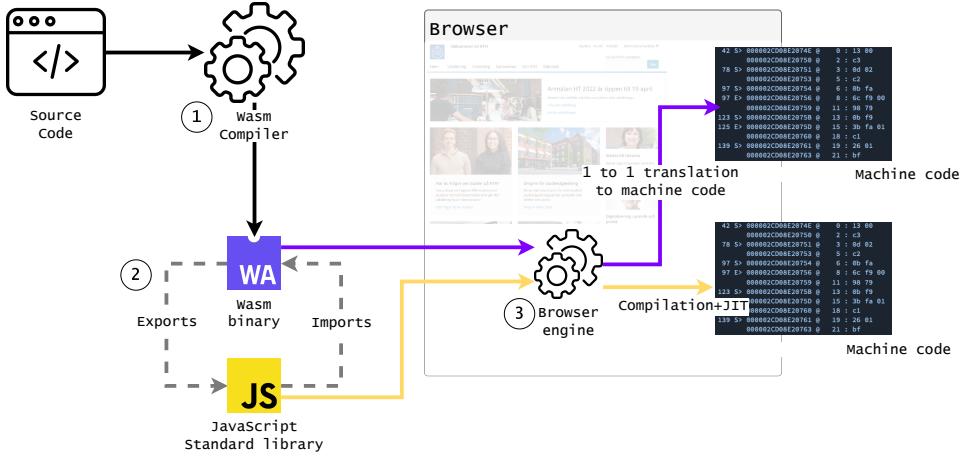


Figure 2.1: WebAssemblybuilding, compilation in the host engine and execution.

Wasm can execute directly and is platform independent. As such it is useful for IoT and Edge computing [5, 21]. For instance, Cloudflare and Fastly adapted their platforms to provide Function as a Service (FaaS) directly with WebAssembly. In this case, the standard library, instead of JavaScript, is provided by any other language stack that the host environment supports. In 2019, the Bytecode Alliance⁶ proposed WebAssemblySystem Interface (WASI) [11]. WASI is the foundation to build Wasm code outside of the browser with a POSIX system interface platform. It standarizes the adoption of WebAssembly outside web browsers [24] in heterogeneous platforms.

2.1.2 WebAssembly specification

WebAssembly defines its own Instruction Set Architecture (ISA) [38]. It is an abstraction close to machine code instructions but agnostic to CPU architectures.

⁶<https://bytecodealliance.org/>

Thus, Wasm is platform independent. The ISA of Wasm includes also the necessary components that the binary requires to run in any host engine. A Wasm binary has a unique module as its main component. A module is composed by sections, corresponding to 13 types⁷, each of them with an explicit semantic and a specific order inside the module. This makes the compilation to machine code faster.

In Listing 2.1 and Listing 2.2 we illustrate a C program and its compilation to Wasm. The C function contains: heap allocation, external function declaration and the definition of a function with a loop, conditional branching, function calls and memory accesses. The code in Listing 2.2 is in the textual format for the generated Wasm. The module in this case first defines the signature of the functions(Line 2, Line 3 and Line 4) that help in the validation of the binary defining its parameter and result types. The information exchange between the host and the Wasm binary might be in two ways, exporting and importing functions, memory and globals to and from the host engine (Line 5, Line 35 and Line 36). The definition of the function (Line 6) and its body follows the last import declaration at Line 5.

The function body is composed by local variable declarations and typed instructions that are evaluated in a virtual stack (Line 7 to Line 32 in Listing 2.2). Each instruction reads its operands from the stack and pushes back the result. The result of a function call is the top value of the stack at the end of the execution. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` at Line 32. A valid Wasm binary should have a valid stack structure that is verified during its translation to machine code. The stack validation is carried out using the static types of Wasm, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. As the listing shows, instructions are annotated with a numeric type.

Wasm manages the memory in a restricted way. A Wasm module has a linear memory component that is accessed as `i32` pointers and should be isolated from the virtual stack. The declaration of the linear data in the memory is showed in Line 37. The memory access is illustrated in Line 15. This memory is usually bound in browser engines to 2Gb of size, and it is only shareable between the process that instantiate the Wasm binary and the binary itself (explicitly declared in Line 33 and Line 36). Therefore, this ensures the isolation of the execution of Wasm code.

Wasm also provides global variables in their four primitive types. Global variables (Line 34) are only accessible by their declaration index, and it is not possible to dynamically address them. For functions, Wasm follows the same mechanism, either the functions are called by their index (Line 30) or using a static table of function declarations. This latter allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, for which the Wasm's compiler is responsible of populating the static table of functions.

In Wasm, all instructions are grouped into blocks, being the start of a function the root block. Two consecutive block declarations can be appreciated in Line 10 and Line 11 of Listing 2.2. Control flow structures jump between block boundaries

⁷<https://webassembly.github.io/spec/core/binary/modules.html#sections>

Listing 2.1: Example C function.

```
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }
    return A[0];
}
```

Listing 2.2: WebAssembly code for Listing 2.1.

```
1  (module
2   (type (;0) (func (param f32) (result i32)))
3   (type (;1) (func))
4   (type (;2) (func (result i32)))
5   (import "env" "ftoi" (func $ftoi (type 0)))
6   (func $main (type 2) (result i32)
7     (local i32 i32)
8     i32.const -1000
9     local.set 0
10    block ;label = @1;
11    loop ;label = @2;
12    i32.const 0
13    local.get 0
14    i32.add
15    i32.load
16    local.tee 1
17    i32.const 101
18    i32.ge_s
19    br_if 1 ;@1;
20    local.get 0
21    i32.const 4
22    i32.add
23    local.tee 0
24    br_if 0 ;@2;
25    end
26    i32.const 0
27    return
28  end
29  f32.const 0x1.9147aep+3
30  call $ftoi
31  local.get 1
32  i32.add)
33  (memory (;0;) 1)
34  (global (;4;) i32 (i32.const 1000))
35  (export "memory" (memory 0))
36  (export "A" (global 2))
37  (data $data (0) "\00\00\00\00...")
38 )
```

and not to any position in the code like regular assembly code. A block may specify the state that the stack must have before its execution and the result stack value coming from its instructions. Inside the Wasm binary the blocks explicitly define where they start and end (Line 25 and Line 28). By design, each block executes independently and cannot execute or refer to outer block values. This is guaranteed by explicitly annotating the state of the stack before and after the block. Three instructions handle the navigation between blocks: unconditional break, conditional break (Line 19 and Line 24) and table break. Each break instruction can only jump to one of its enclosing blocks. For example, in Listing 2.2, Line 19 forces the execution to jump to the end of the first block at Line 10 if the value at the top of the stack is greater than zero.

2.1.3 WebAssembly security

As we described, WebAssembly is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [22] and makes it easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

WebAssembly is vulnerable [30]. Implementations in both browsers and standalone runtimes [5] are vulnerable. Genkin et al. demonstrated that Wasm could be used to exfiltrate data using cache timing-side channels [35]. Moreover, binaries itself can be vulnerable. The work of Lehmann et al. [19] proved that C/C++ source code vulnerabilities can propagate to Wasm such as overwriting constant data or manipulating the heap using stackoverflow. Even though these vulnerabilities need a specific standard library implementation to be exploited, they make a call for better defenses for WebAssembly. Recently, Stiévenart and colleagues demonstrate that C/C++ source code vulnerabilities can be ported to Wasm [1]. Several proposals for extending WebAssembly in the current roadmap could address some existing vulnerabilities. For example, having multiple memories⁸ could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors⁹.

2.2 Software Diversification

Software Diversification has been widely studied in the past decades. This section discusses its state of the art. Software diversification consists in synthesizing, reusing, distributing, and executing different, functionally equivalent programs. According to the survey of Baudry and Monperrus [48], the motivation for software diversification can be separated in five categories: reusability [71], software testing [61], performance [58], fault tolerance [82] and security [79]. Our work contributes to the latter two categories. In this section we discuss related works by highlighting how they generate diversification and how they put it into practice.

There are two primary sources of software diversification: Natural Diversity and Artificial Diversity[48]. This work contributes to the state of the art of Artificial Diversity, which consists of synthesizing software. This thesis is founded on the work of Cohen in 1993 [79] as follows.

⁸<https://github.com/WebAssembly/multi-memory/blob/main/proposals/multi-memory/Overview.md>

⁹<https://webassembly.org/roadmap/>

2.2.1 Variants' generation

Cohen et al. proposed to generate artificial software diversification through mutation strategies. A mutation strategy is a set of rules to define how a specific component of software development should be changed to provide a different yet functionally equivalent program. Cohen et al. proposed 10 concrete transformation strategies that can be applied at fine-grained levels. All described strategies can be mixed together. They can be applied in any sequence and recursively, providing a richer diversity environment. We summarize them, complemented with the work of Baudry et al. **TODO Recheck** [48] and the work of Jackson et al. [56], in 5 strategies.

(S1) Equivalent instructions replacement Semantically equivalent code can replace pieces of programs. This strategy replaces the original code with equivalent arithmetic expressions or injects instructions that do not affect the computation result. There are two main approaches for generating equivalent code: rewriting rules and exhaustive searching. The replacement strategies are written by hand as rewriting rules for the first one. A rewriting rule is a tuple composed of a piece of code and a semantic equivalent replacement. For example, Cleempot et al. [57] and Homescu et al. [55] insert NOP instructions to generate statically different variants. In their works, the rewriting rule is defined as `instr => (nop instr)`, meaning that `nop` operation followed by the instruction is a valid replacement. On the other hand, exhaustive searching samples all possible programs for a specific language. In this topic, Jacob et al. [64] proposed the technique called superdiversification for x86 binaries. The superdiversification strategy proposed by Jacob and colleagues performs an exhaustive search of all programs that can be constructed from a specific language grammar. If one of the generated programs is equivalent to the original program, then it is reported as a variant. Similarly, Tsoupidi et al. [15] introduced Diversity by Construction, a constraint-based compiler to generate software diversity for MIPS32 architecture.

(S2) Instruction reordering This strategy reorders instructions or entire program blocks if they are independent. The location of variable declarations might change as well if compilers re-order them in the symbol tables. It prevents static examination and analysis of parameters and alters memory locations. In this field, Bhatkar et al. [75, 72] proposed the random permutation of the order of variables and routines for ELF binaries.

(S3) Adding, changing, removing jumps and calls This strategy creates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [79] mainly illustrated the case by inserting bogus jumps in programs. Pettis and Hansen [80] proposed to split basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [47] de-inline basic blocks of code as an LLVM pass. In their approach, each de-inlined code is transformed into semantically equivalent functions that are randomly selected at

runtime to replace the original code calculation. On the same topic, Bhatkar et al. [72] extended their previous approach [75], replacing function calls by indirect pointer calls in C source code, allowing post binary reordering of function calls. Recently, Romano et al. [2] proposed an obfuscation technique for JavaScript in which part of the code is replaced by calls to complementary Wasm function.

(S4) Program memory and stack randomization This strategy changes the layout of programs in the host memory. Also, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [75, 72] propose to randomize the base addresses of applications and the library memory regions in ELF binaries. Tadesse Aga and Autin [28] and Lee et al. [6] propose a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [69] propose to separate a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [13] transforms programs to reduce memory exposure time, improving the time needed for frequent memory address randomization.

(S5) ISA randomization and simulation This strategy uses a key to cypher the original program binary into another encoded binary. Once encoded, the program can be decoded only once at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving code inspection. Kc et al. [74] and Barrantes et al. [76] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [77] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [44] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL. Code obfuscation [2] **TODO** *find a new citation here* can be seen as a simplification of *ISA randomization*. The main difference between encoding and obfuscating code is that the former requires the final target to know the encoding key while the latter executes as it is in any client. Yet, both strategies are meant to tackle program analysis from potential attackers.

2.2.2 Variants' equivalence

Equivalence checking between program variants is an essential component for any program transformation task, from checking compiler optimizations [53] to the artificial synthesis of programs discussed in this chapter. Equivalence checking proves that two pieces of code or programs are semantically equivalent [26]. Cohen [79] simplifies this checking by enunciating the following property: two programs are equivalent if given identical input, they produce the identical output. We use this same enunciation as the definition of *functional equivalence* along with this dissertation. Equivalence checking in Software Diversification aims to preserve

the original functionality for programs while changing observable behaviors. For example, two programs can be statically different or have different execution times and provide the same computation.

The equivalence property is often guaranteed by construction. For example, in the case illustrated in S1 for Cleempot et al. [57] and Homescu et al. [55], their transformation strategies are designed to generate semantically equivalent program variants. However, this process is prone to developer errors, and further validation is needed. For example, the test suite of the original program can be used to check the variant. If the test suite passes for the program variant [20], can be considered equivalent to the original. However, it is limited due to the need for a preexisting test suite. When the test suite does not exist, another technique is needed to check for equivalence.

If there is no test suite or the technique does not inherently implement the equivalence property, the previously mentioned works use theorem solvers (SMT solvers) [65] to prove equivalence. For SMT solvers, the main idea is to turn the two code variants into mathematical formulas. The SMT solver checks for counter-examples. When the SMT solver finds a counter-example, there exists an input for which the two mathematical formulas return a different output. The main limitation of this technique is that all algorithms cannot be translated to a mathematical formula, for example, loops. Yet, this technique tends to be the most used for no-branching-programs checking like basic block and peephole replacements [43].

Another approach to check equivalence between two programs similar to using SMT solvers is by using fuzzers [37]. Fuzzers randomly generate inputs that provide different observable behavior. If two inputs provide a different output in the variant, the variant and the original program are not equivalent. The main limitation for fuzzers is that the process is remarkably time-expensive and requires the introduction of oracles by hand.

2.2.3 Usages of Software Diversity

After program variants are generated, they can be used in two main scenarios: Randomization or Multivariant Execution(MVE) [56]. In Figure 2.2a and Figure 2.2b we illustrate both scenarios.

(U1) Randomization: In the context of our work *Randomization* refers to the ability of a program to be served as different variants to different clients. In the scenario of Figure 2.2a, a program is selected from the collection of variants (program's variant pool), and at each deployment, it is assigned to a random client. Jackson et al. [56] compare the variant's pool in Randomization with a herd immunity, since vulnerable binaries can affect only part of the client's community.

El-Khalil and colleagues [73] proposed to use a custom compiler to generate different binaries out of the compilation process. El-Khalil and colleagues modified a version of GCC 4.1 to separate a conventional stack into several component parts,

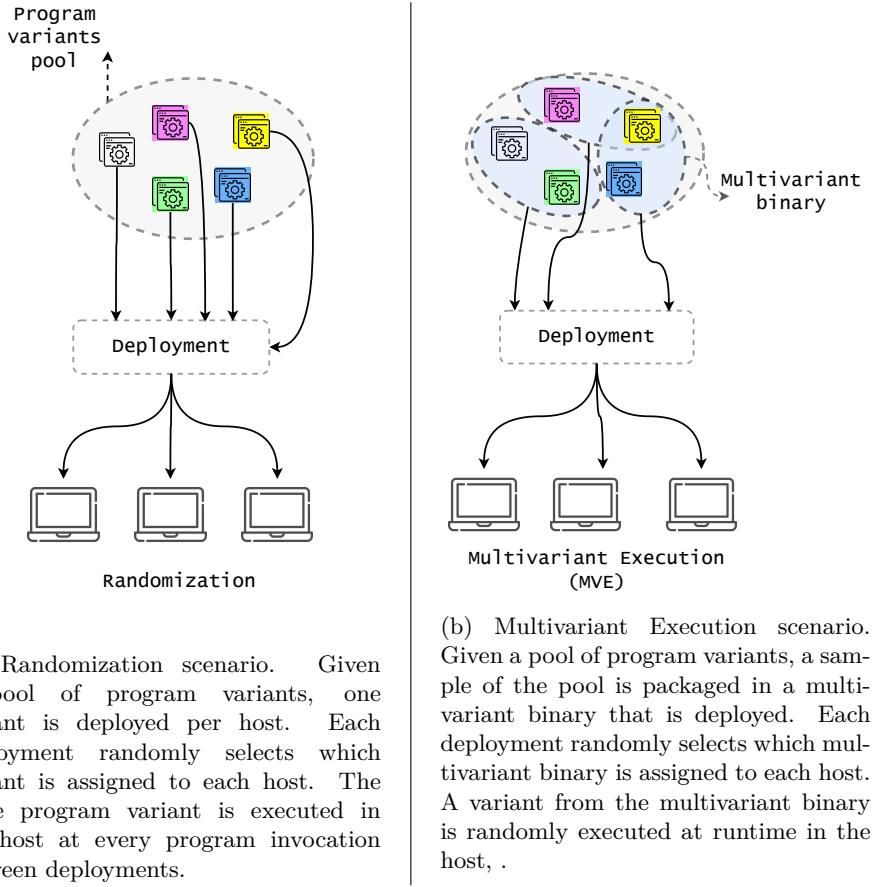


Figure 2.2: Software Diversification usages.

called multistacks. On the same topic, Aga and colleagues [28] proposed to generate program variants by randomizing its data layout in memory. Their approach makes each variant to operate the same data in memory with different memory offsets. Remarkably, the Polyverse company ¹⁰ materialize randomization at the commercial level in real life. They deliver a unique Linux distribution compilation for each of its clients by scrambling the Linux packages at the source code level.

Virtual machines and operating systems can be also randomized. On this topic, Kc et al. [74] create a unique mapping between artificial CPU instructions and real ones. Their approach makes possible the assignation of different variants to specific target clients. Similarly, Xu et al. [13] recompile the Linux Kernel to reduce the exposure time of persistent memory objects, increasing the frequency of address

¹⁰<https://polyverse.com/>

randomization.

(U2) Multivariant Execution (MVE): In the second scenario Figure 2.2b, multiple program variants are composed in one single binary (multivariant binary) [70]. Each multivariant binary is randomly deployed to a client. Once in the client, the multivariant binary executes its embedded program variants at runtime.

The execution of the embedded variants can be either in parallel to check for inconsistencies or a single program to randomize execution paths [75]. Bruschi et al. [68] extended the idea of executing two variants in parallel with not-overlapping and randomized memory layouts. Simultaneously, Salamat et al. [66] modified a standard library that generates 32-bit Intel variants where the stack grows in the opposite direction, checking for memory inconsistencies. Notably, Davi et al. proposed Isomeron [46], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. The previously mentioned works showed the benefits of exploiting the limit case of executing only two variants in a multivariant environment. Agosta et al. [50] and Crane et al. [47] used more than two generated programs in the multivariant composition, randomizing software control flow at runtime.

Both scenarios have demonstrated to harden security by tackling known vulnerabilities such as (JIT)ROP attacks [59] and power side-channels [60]. Moreover, Artificial Software Diversification is a preemptive technique for yet unknown vulnerabilities [56]. Our work contributes to both usages scenarios for WebAssembly.

2.3 Open challenges

In Table 2.1 we list the related work on Artificial Software Diversification that support our approaches. The first and second columns in the table correspond to the author names and the references to their work, followed by one column for each strategy and usage (S1, S2, S3, S4, S5, U1 and U2). The last column of the table summarizes the technical contribution and the reach of the referred work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order.

Our first contribution, CROW [10] generates multiple program variants for WebAssembly using the LLVM pipeline. It contributes to state of the art in artificially creating randomization for WebAssembly (U1). Due to the specificities of code execution in the browser (mentioned in Section 2.1), this can be considered as a randomization approach. For example, since WebAssembly is served at each page load, every time a user asks for a WebAssembly binary, she can be served a different variant provided by CROW. With MEWE [9], our second contribution,

we randomly select from several variants at runtime, creating a multivariant execution scheme(U2) that randomizes the observable behaviors at each run of the multivariant binary.

Conclusions

In this chapter, we presented the background on the WebAssembly language, including its security issues and related work. This chapter aims to settle down the foundation to study automatic diversification for WebAssembly. We highlighted related work on Artificial Software Diversification, showing that it has been widely researched, not being the case for WebAssembly. On the other hand, current available implementations for Software Diversification cannot be directly ported to Wasm. We placed our contributions in the field of artificial diversity. In Chapter 3 we describe the technical details that lead our contributions. Besides, the impact of our contributions is evaluated by following the methodology described in Chapter 4.

Authors	S1	S2	S3	S4	S5	U1	U2	Main technical contribution
Pettis and Hansen [80]		✓		✓		✓		Custom Pascal compiler for PA-RISC architecture
Chew and Song [77]			✓			✓		Linux Kernel recompilation.
Kc et al. [74]					✓			Linux Kernel recompilation.
Barrantes et al. [76]				✓		✓		x86 to x86 transformations using Valgrind
Bhatkar et al. [75]	✓	✓		✓		✓		ELF binary transformations
El-Khalil and Keromytis [73]						✓		custom GCC compiler for x86 architecture
Bhatkar et al. [72]	✓	✓		✓		✓		C/C++ source to source transformations and ELF binary transformations
Younan et al. [69]					✓			custom GCC compiler
Bruschi et al. [68]					✓		✓	ELF binary transformations.
Salamat et al. [66]			✓			✓		Custom GNU compiler
Jacob et al. [64]	✓	✓						x86 to x86 transformations
Salamat et al. [62]					✓		✓	x86 to x86 transformations
Amarilli et al. [60]	✓			✓		✓		Polymorphic code generator for ARM architecture
Jackson [56]	✓				✓	✓		LLVM compiler, only backend for x86 architecture
Cleemput et al. [73]	✓				✓			x86 to x86 transformations
Homescu et al. [55]	✓				✓			LLVM 3.1.0 [†]
Crane et al. [47]	✓	✓	✓			✓		LLVM, only backend for x86 architecture
Davi et al. [46]						✓		Windows DLL instrumentation
Couroussé et al. [44]	✓	✓			✓	✓		Custom GCC compiler targeting microcontrollers
Lu et al. [31]				✓		✓		GNU assembler for Linux kernel
Belleville et al. [36]	✓			✓		✓		Only C language frontend, LLVM 3.8.0 [†]
Aga et al. [28]				✓		✓		Data layout randomization, LLVM 3.9 [†]
Österlund et al. [25]				✓		✓		Linux Kernel recompilation.
Xu et al. [13]				✓		✓		Custom kernel module in Linux OS
Lee et al. [6]				✓		✓		LLVM 12.0.0 backend for x86

[†] Notice that LLVM only supports WebAssembly backend from version 8.0.0

Table 2.1: The first and second columns in the table correspond to the author names and the references to their work, followed by one column for each strategy and usage (S1, S2, S3, S4, S5, U1 and U2). The last column of the table summarizes the technical contribution and the reach of the referred work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order. The last two rows locate our contributions.

